Project Report: Digit Recognizer

Jimmy Lin (xl5224)
Department of Computer Science
University of Texas at Austin
Austin, TX 78712
`JimmyLin@utexas.edu`

Jisong Yang (jy7226)
Department of Statistical Scientfic Computation
University of Texas at Austin
Austin, TX 78712
`jsyang993@gmail.com`

Abstract

In this project proposal, we derived several models (PCA-KNN, Neural Network, IDM) that are able to recognize digits with a decent accuracy. And their advantages and drawbacks are compared. To make the project more interesting, we participated in Digits Reconizer competition, organized in Kaggle to test our models. We attempted the simplest deformation model called IDM and figure out the computational solution through distributed and concurrent system.

1 Introduction

1.1 Problem Description

Recognizing hand-written characters has long been a hot topic in artificial intelligence community. Handwriting recognition is the ability of a computer to receive and interpret intelligible handwritten input from sources such as paper documents, photographs, touch-screens and other devices. In the past decades, many probabilistic or non-probabilistic algorithms have been exploited to effectively solve this problem, including K-nearest neighbour, random forest, and well-known neural network.

1.2 Motivations

As to the historical application of handwriting recognition, it can be traced back to early 1980s. There are plenty of commercial products incorporating handwriting recognition as a replacement for keyboard input were introduced. The hardware application continued to develop in the following decades. Until the recent years, Tablet PCs can be regarded as a special notebook computer that is outfitted with a digitizer tablet and a stylus, and allows a user to handwrite text on the unit's screen. In addition, highly efficient handwriting recognizers were developed in real world to apply on the zip codes detection.

Figure 1: Digit Recognition Examples

Plenty of mature solutions has already been developed to existence in solving the handwriting problem. Since 2009, the recurrent neural networks and deep feedforward neural networks developed in the research group of Jrgen Schmidhuber at the Swiss AI Lab IDSIA have outshined many other models in competitions. The fact that existing algorithms are powerful in recognizing digits does not remove possibility of further improvement. The chance of improvement lies in reduction of computational complexity for recognizing digits in higher degree.

1.3 Dataset

As mentioned previously, the dataset provided by the holder of digit recognition kaggle competition came from the famous MNIST dataset. The **MNIST database (Mixed National Institute of Standards and Technology database)** is a large database of handwritten digits that is commonly used for training various image processing systems. It was created by "re-mixing" the samples from NIST's original datasets. The creators felt that since NIST's training dataset was taken from American Census Bureau employees, while the testing dataset was taken from American high school students, NIST's complete dataset was too hard. Furthermore, the black and white images from NIST were normalized to fit into a 20x20 pixel bounding box and anti-aliased, which introduced grayscale levels.

The competition holder provides us two csv data files, train.csv and test.csv. The data files train.csv and test.csv contain gray-scale images of hand-drawn digits, from zero through nine. Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255, inclusive.

The training data set (in train.csv), has 42,000 rows and 785 columns. The first column, called "label", is the digit that was drawn by the user. The rest of the columns contain the pixel-values of the associated image. Each pixel column in the training set has a name like pixelx, where x is an integer between 0 and 783, inclusive. To locate this pixel on the image, suppose that we have decomposed x as $x = i \times 28 + j$, where i and j are integers between 0 and 27, inclusive. Then pixelx is located on row i and column j of a 28×28 matrix (indexing by zero).

$$\begin{array}{ccccccc} 000 & 001 & 002 & 003 & \ldots & 026 & 027 \\ 028 & 029 & 030 & 031 & \ldots & 054 & 055 \\ 056 & 057 & 058 & 059 & \ldots & 082 & 083 \\ \vdots & \vdots & \vdots & \vdots & \ldots & \vdots & \vdots \\ 728 & 729 & 730 & 731 & \ldots & 754 & 755 \\ 756 & 757 & 758 & 759 & \ldots & 782 & 783 \end{array} \quad (1)$$

The test data set (in test.csv), is the same as the training set, except that it does not contain the "label" column and the test data has 28,000 images in total.

1.4 Report Organization

In this report, we will illuminate the motivation of digit recognition problem on section 1.2. In section 2, the main models exploited in this project will be sequentially present to reader. Those algorithms are PCA-KNN, Neural Network, Multi-class SVM and Deformation Model. For each model, the algorithmic description, the motivation of attempting that algorithm and finally relevant issues on practical implementation will be indicated.

2 Implementation Details

2.1 PCA-KNN

Principal Component Analysis (PCA) is a statistical procedure that uses orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. Typically, the number of principal components is no more than the number of original variables. That is to say, a particular subspace, whose bases are independent to each other, is generated by such tranformation. This transformation is defined in such a way that the first principal component has the largest possible variance, and each succeeding component in turn has the highest variance possible under the constraint that it is orthogonal to the preceding components. In section 3, the performance comparisons of different algorithms will be illustrated and after which, we will make conclusion about which one achieve the best performance.

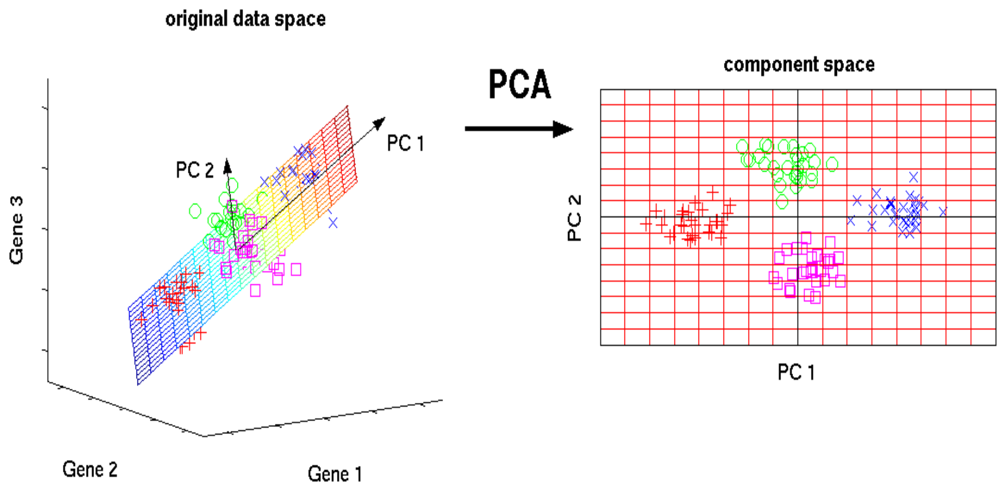


Figure 2: Principle Component Analysis

In our project, we make use of PCA as dimensionality reduction for preprocessing the raw input image data. Since the raw data explictly characterize the gray value of each pixels, each handwritten instance is extremely high-dimensional ($28 \times 28 = 784$ in total). Although directly using the raw input data avoids loss of information, the computational complexity would be unresonably large. Put it another way, we have to think of a way to compromise the ultimate accuracy to achieve a less computational cost on that algorithm.

However, several details matter significantly to the effect of PCA. One is the number of principal components employed. If too many principal components are used, it is far from reaching the goal of dimensionality reduction. However, we may lose useful information for latter prediction if important components are not used. Note that even least useful principal components contain disjoint information that may contribute to latter prediction performance. As to our inplementation, we cross validate on the number of principal components, along with the number of voting neighbours k.

k-Nearest Neighour is a well-known non-parametric method used for classification and regression. In k-NN classification, the output is a class membership. An object is classified by a majority vote of its k neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). In the particular case of $k = 1$, the object is simply assigned to the class of that single nearest neighbor. Another characteristic of KNN is its lazyness. Its objective function for learning is only approximated locally and all computation is deferred until the stage of classification.

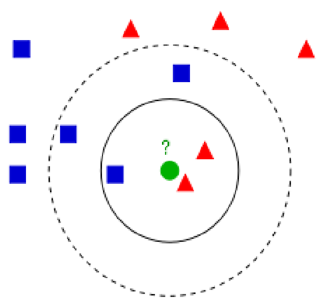


Figure 3: k-Nearest Neighour

Arguments supporting the usage of k-Nearest Neighour are tremendously dominant in the task of digit recognition. The first and most significant reason lies in the the simplicity of its implementation. Another dominant factor is that the k-Nearest Neighour has decent performance (prediction accuracy) if given sufficent amount of training data. Hence, we cannot reject the temptation of using k-Nearest Neighour as a baseline classifier.

However, the weakness of k-Nearest Neighour lies in its vulnerability to badly scaled dimension when the distance between arbitrary data objects is computed. This is especially true when the input data has extremely large number of features. To solve this problem, we apply min-max normalization after the preprocessing step. Another issue we concerned about is the determination of the range of voting. In other word, how many neighbours are allowed to participate in the decision process of one object's class membership? As stated previously, we apply the classic technique Cross Validation to determine the parameter k so as to reach the optimal performance.

2.2 Neural Network

In the field of machine learning, **neural networks** are computational models inspired by animals' central nervous systems (in particular the brain) that are capable of machine learning and pattern recognition. They are usually presented as systems of interconnected "neurons" that can compute values from inputs by feeding information through the network. Another name for neural network is artificial neural network (ANN).

The graphical representation of variables are as follows. The nodes in input layer represents the input features of each training entity. Those nodes in hidden layer corresponds to the learned features, served for final regression or classification decision of that entity. And the only node in output layer corresponds to the regression or prediction decision.

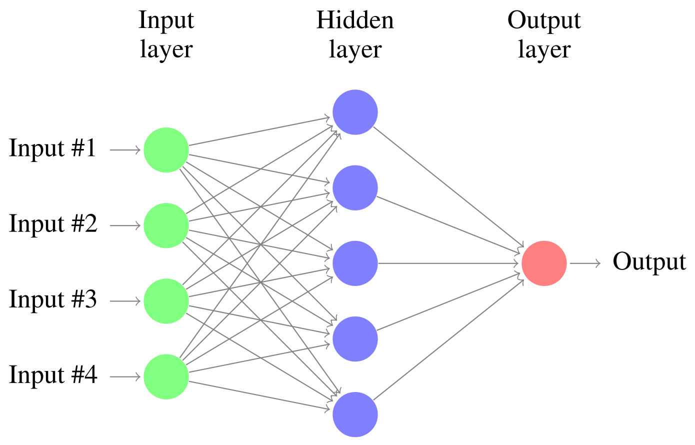


Figure 4: Neural Network Diagram

An ANN is typically defined by three types of parameters: (a) The interconnection pattern between the different layers of neurons (b) The learning process for updating the weights of the interconnections (c) The activation function that converts a neuron's weighted input to its output activation.

Mathematically, a series of training entities are provided. We just take one entity $\mathbf{x}$ as example. The entire input layer are fed by the instance $\mathbf{x} = (x_1, ..., x_n)$. And the information passed in to the j-th hidden nodes a_j is computed by

$$a_j = \sum_i w_{ji} x_i \tag{2}$$

For each hidden node, after receiving the information from input layer, derive its activated value y_j through what it acquire by activation function $h(x)$:

$$y_j = h(a_j) = h(\sum_i w_{ji} x_i) \tag{3}$$

Since we formulate digit recognition as a multi-class classification problem, the activation function in the output layer should be a generalized logistic function, that is, softmax function. Here, we provide the definition of softmax function as follows: given a set of

$$\sigma(\mathbf{z}) = \frac{e^{\mathbf{z}_j}}{\sum_{k=1}^{K} e^{\mathbf{z}_k}} \tag{4}$$

Note that the input vector $\mathbf{z}$ should be offset down by maximum element of that vector, in which way the ”blowup” of every individual exponential term can be avoided. This is actually a common problem in the scientific computing.

To train neural network model, we need to feed the neural network with a series of entities. The flow of information, while doing training, involves in two stages: forward propagation and backward propagation. In the stage of forward propagation, the network accept a new training entity at input layer , pass the value forward until output layer and finally evaluate the prediction over that particular entity. Since the error (difference between prediction label and groudtruth label) was derived, it comes to the stage of the back propagation. In this stage, the pre-computed error was passed back towards the input layer so as to update the parameters (weight matrix). We can keep on feeding training entity to this neural network until convergence.

The stage of backward propagation is much more complicated. On the output layer, the error is simply as the difference of output and activated value on that unit. However, the error of hidden nodes are computed similar to the propagation in the first forward stage. Formally, the error δ_j of node indexed by j in previous layer can be represented as

$$\delta_i = h'(a_j) \sum_k w_{kj} \delta_k \tag{5}$$

Every nodes on one hidden layer first collects all error from next layer, summarize them as its error made by itself and finally update the weight matrix. Note that there is no error for neurons on input layer.

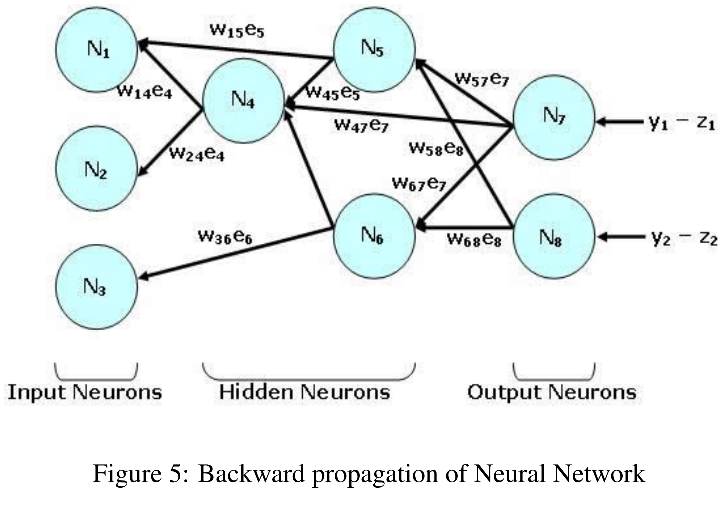


Figure 5: Backward propagation of Neural Network

As to the motivation of using neural network, one most significant factor comes to the fact that Neural Networks are usually employed as one universal function approximator. Formally, if given sufficent hidden nodes, the neural network model can approximate any function to arbitrary precision. Ituitively, we can treat the digit number as a function of a series features. In original setting, these features are gray value of every pixel. If preprocessed by PCA, the features become the principal components chosen by us.

There are also several concerns for the implementation of Neural Network model. The first one coming to our mind is the number of hidden nodes for usage. We turned to various experiment report for the parameter optimization of neural network on digit recognition. It turns out that about five to six layers of nodes will result in astonishingly good performance. One convincing approach to address this problem is to cross validate on the network structure, but this requires a large computational cost. Another concern is the determination of leraning rate α. The third one is selection of activation function for hidden nodes. Typically, there are a few options: $tan(x)$, $log(x)$, $exp(x)$. We can cross validate over all these activation function for nodes of each layer. Last but not in the least, how to avoid overfitting remained a challenging issue. Theoretically, early stopping and fine-tune regularization parameters are both suitable approaches to avoid overfitting of neural network. However, our attempted implementation does not incorporate any of them.

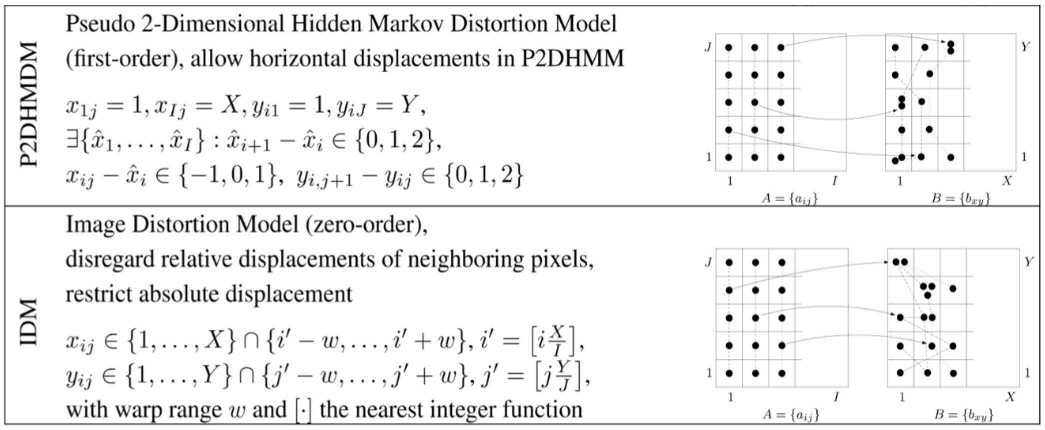
2.3 Deformation Model

In academia, the deformation models are especially suited for local changes as they often occur in the presence of image object variability. And among the advanced models like Convolutional Neural Network, Deep Belief Network, the deformation model is one approach that combines simplicity of implementation, low-computational complexity, and highly competitive performance across various real-world image recognition tasks.

The intuitive idea of deformation model is rather simple. Given a test image A and an series of test image B, the category of one object is determined by the closest reference image. However, the concepts of ”distance” is not simply euclidean distance. It is euclidean distance over the non-linear mapping feature space. The decision rule is formulated by the following formula:

$$k(A) = \underset{k}{\operatorname{argmin}}\{\min_{n=1,..,N_k} d(A, B_k n)\} \tag{6}$$

The collections of existing deformation model identified with its constraints imposed on the non-linear image deformation mapping are shown in the following Figure.



P2DHMDM	Pseudo 2-Dimensional Hidden Markov Distortion Model (first-order), allow horizontal displacements in P2DHMM $x_{1j} = 1, x_{Ij} = X, y_{i1} = 1, y_{iJ} = Y,$ $\exists\{\hat{x}_1, \ldots, \hat{x}_I\} : \hat{x}_{i+1} - \hat{x}_i \in \{0, 1, 2\},$ $x_{ij} - \hat{x}_i \in \{-1, 0, 1\},\ y_{i,j+1} - y_{ij} \in \{0, 1, 2\}$	
IDM	Image Distortion Model (zero-order), disregard relative displacements of neighboring pixels, restrict absolute displacement $x_{ij} \in \{1, \ldots, X\} \cap \{i' - w, \ldots, i' + w\}, i' = \left[i\frac{X}{I}\right],$ $y_{ij} \in \{1, \ldots, Y\} \cap \{j' - w, \ldots, j' + w\}, j' = \left[j\frac{Y}{J}\right],$ with warp range w and $[\cdot]$ the nearest integer function	

Figure 6: Constraints of different Deformation Mappings for different model

Our implementation only involves in the simplest deformation model, the one with zero-order non-linear mapping called IDM (Image Distortion Model). The algorithmic description is shown in Figure 7.

Even though Image Distortion Model is identified as the least computational complexity among all deformation models and the advanced machine learning algorithms, it still requires parallel computing to bring the result to light within two days. In fact, we self-design the distributed computing architecture with the employment of about 50 UTCS Linux machine and four 3.3GHZ CPU on each

```
Algorithm 3 IDM-distance; input: test image $A$,
reference image $B$;
for $i = 1$ to $I$
    for $j = 1$ to $J$
        $i' = \left[i \frac{X}{I}\right], j' = \left[j \frac{Y}{J}\right]$
        $s = s + \min_{\substack{x \in \{1,\ldots,X\} \cap \{i'-w,\ldots,i'+w\} \\ y \in \{1,\ldots,Y\} \cap \{j'-w,\ldots,j'+w\}}} \|a_{ij} - b_{xy}\|^2$
output: $s$
```

Figure 7: Algorithmic description of Image Distortion Model

machine. For further detailed implementation of the distributed computing framework, please see the attachment of the report.

3 Result

3.1 Commitment History

Progress #1 issued at March 29

1. Jimmy Lin: We work out initial version of **Principle Component Analysis**(PCA), and submit it with KNN classification model. In this submission, we acquire an prediction accuracy of **0.96886** and ranked at **148** at that date.

2. Jimmy Lin: We also tried Random Forest method based on PCA 100 pricipal components and achieve **0.95457** accuracy.

Progress #2 issued at April 14

1. Jimmy Lin: C-SVM with preprocessed data by **Principle Component Analysis** (100 principal components). The prediction accuracy achieved is **0.91043**.

2. Jimmy Lin: NU-SVM with the same configuration achieves accuracy **0.89743**.

3. Jimmy Lin: code up our own implementation of neural network. However, our implementation does not come out with satisfyingresult, perhaps because of no mechanism to avoid overfitting.

Progress #3 issued at April 16

1. Jimmy Lin: Reference to libsvm's C-SVM method with normalization preprocessing achieves **0.93571**.

2. Jimmy Lin: Implement Majority Voting Ensemble Method, containing PCA-KNN, KNN, libsvm-All-CSVM, achieves **0.96714**. Still no breakthrough.

Progress #4 issued at April 24

1. Jason Yang: Apply Cross Validation Techniques to determine the optimal parameters of PCA-KNN model. The prediction accuracy achieved is **0.97300** and in the meanwhile, we refresh our ranking to 99.

Progress #5 issued at April 26

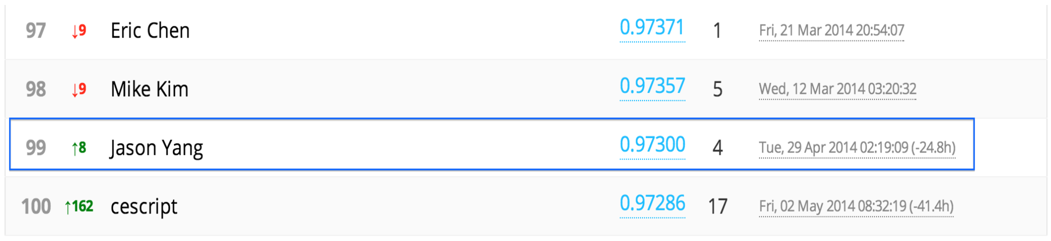
1. Jason Yang: Turn to the implementation of neural network implementation in Deep Learning Toolbox. And achieves an accuracy of ** **.

Progress #6 issued at April 27

1. Jimmy Lin: read the paper of deformation model and code up for it. To compute the solution by this model, we make use of over 50 UTCS Linux machine to run in parallel. However, our implementation only achieves **0.92629**, which does not reach our expectation.

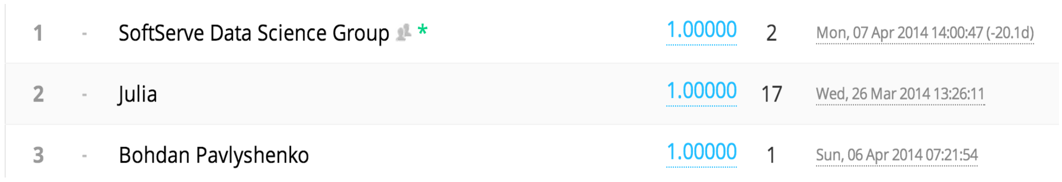
3.2 Ranking on Kaggle Competition

Our optimal solution model **PCA-KNN** ranks at 99 of the leader board. Following is the snapshot of our position on that leaderboard.



97	↓9	Eric Chen	0.97371	1	Fri, 21 Mar 2014 20:54:07
98	↓9	Mike Kim	0.97357	5	Wed, 12 Mar 2014 03:20:32
99	↑8	Jason Yang	0.97300	4	Tue, 29 Apr 2014 02:19:09 (-24.8h)
100	↑162	cescript	0.97286	17	Fri, 02 May 2014 08:32:19 (-41.4h)

Figure 8: Ranking of our best solution (PCA-KNN model)



1	-	SoftServe Data Science Group *	1.00000	2	Mon, 07 Apr 2014 14:00:47 (-20.1d)
2	-	Julia	1.00000	17	Wed, 26 Mar 2014 13:26:11
3	-	Bohdan Pavlyshenko	1.00000	1	Sun, 06 Apr 2014 07:21:54

Figure 9: Performance of top solution in this competition

For further details about this leader board, please visit

`http://www.kaggle.com/c/digit-recognizer/leaderboard`

3.3 More about Optimal Solution: PCA-KNN

Our goal is applying eigenvector decomposition of covariance (also known as PCA) for identifying the images. First, eigenvectors with larger eigenvalues are extracted from the sample trainingImages to retain the principal components of the images. Second, projecting both testImages(28000) and trainImages(42000) onto the eigenspace, which is reduced dimension. Finally, using the features captured by projection to identify testImages by K nearest neighbor classification. In order to capture the principal components of the images, the eigenvectors derived from the samples of trainImages are sorted from the largest to the smallest according to their eigenvalues. The top T eigenvectors are selected because they represent in which directions the data span dramatically. With those principal eigenvectors, less information is need to reconstruct the testimage and identification.

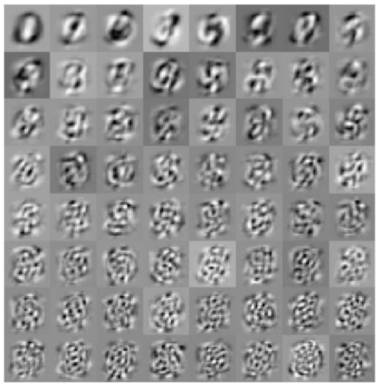


Figure 10: Top 64 Eigenvectors

Another issue is about how many eigenvectors to take. Fig.1 shows the eigenvectors extracted from the trainImages. The sample size N =630, K =5. The top 64 eigenvectors are displayed according to their eigenvalues. The top 8 eigenvectors are much vivid than others. This could be explained that images don't span uniformly in the space, they vary tremendously in some directions, and little in some other directions. By extracting the top eigenvectors, it allows us to capture enough features to reconstruct images and identifying them.

The eigenvectors are calculated in such way: if the sample size N is larger than the X (the total number of pixels of one image), the eigenvector are calculated directly, else if the sample size N is smaller than the X, calculate the eigenvector of size(N*N) first, then derive the original ones by A* V. (Please refer to the code) Then the top eigenvectors are extracted. I take the sample size N from 200 to 40000. Fig.2 shows the outcome. The error rate decreases dramatically from 0.0738 to 0.0351 as the sample size increase and stays around 0.25 after size increases up to 630. Theoretically, the more sample size we have, the better outcome we achieved. But that is not the case. The reason might come from as following: The more features are captured when sample size is increased at the beginning. However, since PCA is linear transformation that map the images from the high dimension into lower dimension, some nonlinear features of the images will not be transformed. That limits PCA's performance.

Cross-validation of Number of sample size for PCA technique

To explore the impact of number of sample on the accuracy, three-nested for-loop is executed. Here is the piece of code:

```
%implement cross-validation
for N = 200: 10: 40000
    for T =4: 2: 512
        for K = 2: 15
            err = crossval(N, T, K, traindat, testdat);
            arrerr2 = [arrerr2 err];
            savefile = 'arrerr2.mat';
            save(savefile, 'arrerr2');
        end
    end
end
```

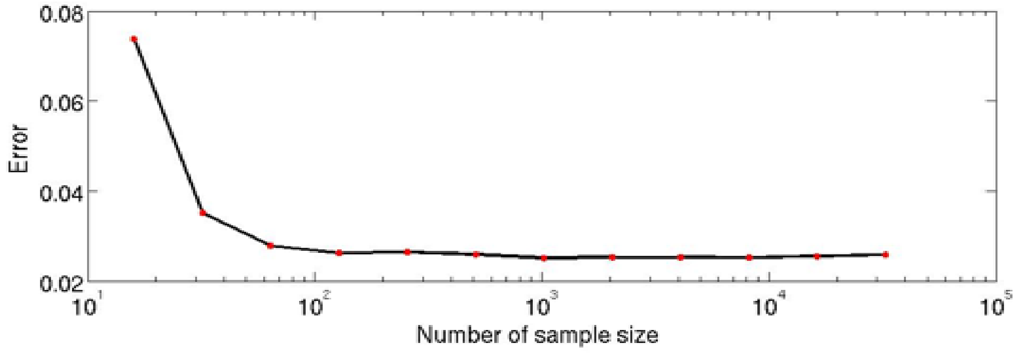


Figure 11: Cross Validation on number of Sample Size used in PCA

Cross-validation of Number of top eigenvectors

Fig.12 shows the impact of number of top eigenvectors on the accuracy. I calculate the error as number of top eigenvectors T varies from 4 to 512. The error drops greatly from 0.36 to 0.25 as T increases from 4 to 64 and does not change too much after that. This verified that using top 64 eigenvectors is good enough to capture the features of images and reconstruction. In addition, it saves us a lot of time. Using all eigenvectors cost 449.59 seconds to finish the test, while using top 64 eigenvectors only need 41.30 seconds

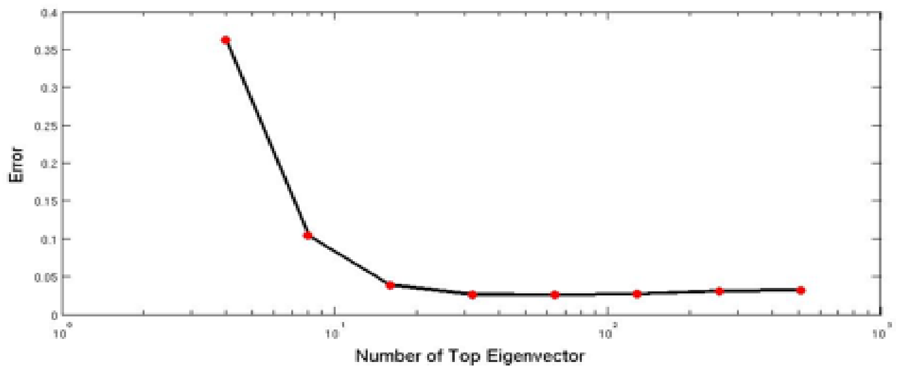


Figure 12: Cross Validation on Number of K

Cross-validation of Number of K

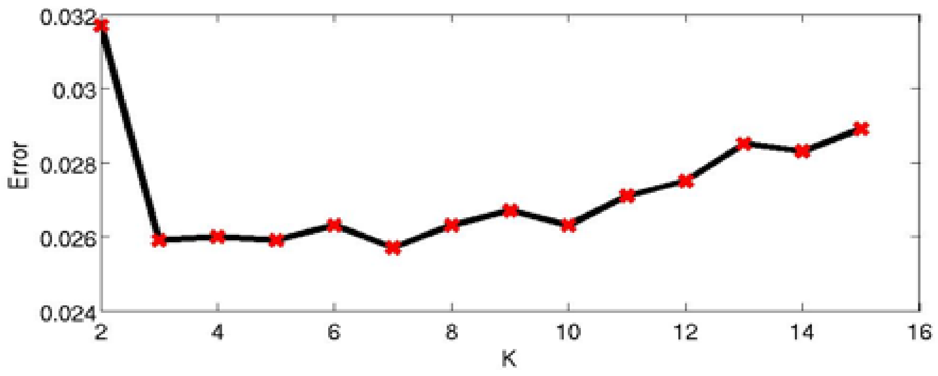


Figure 13: Cross Validation on Number of K

Fig. 13 indicates the effect of K. For each testImage, the nearest K neighbors are searched in the trainImages. I used pdist2 function to compute the euclidean distance between testImage and trainImage. The mode is used to find the most likely label in trainLabels which is compared to the testImages. In my simulation, the optimal value of K is around 5. It is interesting that error increases as K becomes larger. The reason behind might comes from Euclidean distance. The euclidean distance measures the geometrical distance which might not be proper. Take a spiral for example, Euclidean distance would find the wrong neighbor. The more neighbors are found, the more interferences are introduced.

4 Conclusion and Discussion

High model complexity does not indicate high performance. It is true that the academia has developed plenty of theoretically elegant model, most of which are difficult to implement and understand. However, they are not necessarily suitable for any problem on any circumstance. In our experiment, the PCA-KNN model is simplest in both theoretical and practical respects, but it achieves highest performance for (maybe particularly) this digit recognition problem.

Sometimes, the quality and quantity of the dataset matters a lot. It has been a long debate over whether the dataset plays more significant role over the model in the area of data mining and machine learning or whether model is supposed to be more important. It can be extremely hard to say one is always over the other. Nevertheless, from our project, we can say that sometimes the quantity really does matter, and even take a dominant role. The observation supporting this argument is that the k-Nearest Neighour model does not perform well when the given training data is of small size, comparing to the case of large size input like 42,000 training images.

When it comes to the possible further improvement, the first come to our mind is to attempt to make full use of the neural network architecture. Just as stated in the implementation details of neural network, we can actually cross validate on various parameters, including the network structure, activation function for each layer, and the regularization parameter (or iteration number if early stopping technique is used). It is true that the searching space would be rather large if we consider to optimize over such many factors. The way to practically touch the limit of neural network model in digit recognition problem is to make use of our existing distributed computing framework.

5 References

[1] Liu, C. L., Nakashima, K., Sako, H., & Fujisawa, H. (2003). Handwritten digit recognition: benchmarking of state-of-the-art techniques. *Pattern Recognition*, **36(10)**, 2271-2285.

[2] Le Cun, B. B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1990). Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems.*

[3] LeCun, Y., Jackel, L. D., Bottou, L., Brunot, A., Cortes, C., Denker, J. S., & Vapnik, V. (1995, October). Comparison of learning algorithms for handwritten digit recognition. In *International conference on artificial neural networks* **(Vol. 60)**.

[4] Hinton, G. E., Osindero, S., & Teh, Y. W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, **18(7)**, 1527-1554.

[5] Ciresan, D. C., Meier, U., Gambardella, L. M., & Schmidhuber, J. (2010). Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, *22(12)*, 3207-3220.

[6] Keysers, Daniel, et al. "Deformation models for image recognition." *Pattern Analysis and Machine Intelligence, IEEE Transactions* on 29.8 (2007): 1422-1435.

A Matlab Implementation of PCA-KNN Classifier

A.1 Driver Script

```
% main function to drive the program
%
%   Input: N: an array of number of sample size
%          T: an array of number of princeple of eigenvalue to choose
%          K: number of Nearest neighbor
%   Output: array of error based on different parameter configuration
%
clear; clc; close all;
traindat = csvread('train.csv',1,0);
testdat = csvread('test.csv',1,0);
savefile = 'traindat.mat';
save(savefile, 'traindat');
savefile = 'testdat.mat';
save(savefile, 'testdat');

load 'traindat.mat';
load 'testdat.mat';
K = 5;

%================================================================
arrerr2 = [];   %  explore the effact of number of sample size
% iter = floor(log2(60000));

for N = 200: 10: 1000
    for T =20: 2: 50
        for K = 4: 10
            err = crossval(N, T, K, traindat, testdat);
            arrerr2 = [arrerr2 err]
            savefile = 'arrerr2.mat';
            save(savefile, 'arrerr2');
        end
    end
end

N = 630; T=50; K = 5;
estimate_labels(N, T, K,traindat, testdat);
```

A.2 PCA-KNN Routine

```
function  estimate_labels( N, T, K,traindat, testdat)
%ESTIMATE Summary of this function goes here
%   Detailed explanation goes here
    trainLabels = traindat(:,1);
    A = traindat(:,2:785);
    B = testdat;

    B = B'; A = A';
    [~,trainK] = size(A);
    [~,testK] = size(B);

    sample_A = zeros(784, N);
    ridxarr = randperm(trainK);
    for j = 1 : N

        random_int  = ridxarr(j);
        sample_A(:,j) = A(:,random_int);
    end

    [~, V3] = FindEigendigits(sample_A);
    P = V3(:, 1:T);
    P = P';


    temp_ave = mean(A,2);
    A_mean = double(A - repmat(temp_ave, 1, trainK));

    trainmatrix = P * A_mean;

    temp_ave = mean(B,2);
    B_mean = double(B - repmat(temp_ave, 1, testK));

    testmatrix = P * B_mean;

    [~,Index] = pdist2(trainmatrix',testmatrix','euclidean','Smallest',K);


    est_labels = ones(1, testK)*(-1);
    for k = 1 : testK
        label = trainLabels(Index(:, k));
        est_labels(k) = mode(double(label));
    end


    est_labels = [(1:testK)' est_labels'];
    csvwrite('digit.csv',est_labels);



end
```

A.3 Cross Validation

```
function err = crossval(N, T, K,traindat, ~)
%
%   Input:
%       N: number of sample size we take
%       T: number of Eigenvalue and Eigenvector we take
%       K: number of Nearest neighbor
%   Output:
%       err: error
%
%   Usage: err = MyExperiment(N, T, K)
%


if nargin < 1
    N = 100;
end
if nargin < 2
    T = 100;
end
if nargin < 3
    K = 10;
end

err = [];

trainLabels_org = traindat(:,1);

A = traindat(:,2:785);

% implement cross validation. Ten fold
for i = 1: 10
```

```
    B = A((i-1)*4200+1: i*4200, :);
    testLabels = trainLabels_org((i-1)*4200+1: i*4200,1);

    A1 = A;  A1((i-1)*4200+1: i*4200,:) = [];

    trainLabels = trainLabels_org;
    trainLabels((i-1)*4200+1: i*4200,:) = [];

    B = B'; A1 = A1';

[~,trainK] = size(A1);
[~,testK] = size(B);



sample_A = zeros(784, N);
ridxarr = randperm(trainK);
for j = 1 : N

    random_int  = ridxarr(j);
    sample_A(:,j) = A1(:,random_int);
end

[~, V3] = FindEigendigits(sample_A);
P = V3(:, 1:T);
P = P';


temp_ave = mean(A1,2);
A_mean = double(A1 - repmat(temp_ave, 1, trainK));

trainmatrix = P * A_mean;

temp_ave = mean(B,2);
B_mean = double(B - repmat(temp_ave, 1, testK));

testmatrix = P * B_mean;

[~,Index] = pdist2(trainmatrix',testmatrix','euclidean','Smallest',K);


est_labels = ones(1, testK)*(-1);
for k = 1 : testK
    label = trainLabels(Index(:, k));
    est_labels(k) = mode(double(label));
end


err = [err length(find(est_labels' ~= testLabels)) / length(testLabels)];

% plot the graph, set as comment to prvent slowing the program
%=====================plot the reconstruction images===========
% subplot (2,3,1), imshow(reshape(B(:,3),28,28));
% subplot (2,3,4), imshow(reshape((pinv(P)*testmatrix(:,3)),28,28));
% subplot (2,3,2), imshow(reshape(B(:,2),28,28));
% subplot (2,3,5), imshow(reshape((pinv(P)*testmatrix(:,2)),28,28));
% subplot (2,3,3), imshow(reshape(B(:,1),28,28));
% subplot (2,3,6), imshow(reshape((pinv(P)*testmatrix(:,1)),28,28));

% subplot (2,3,1), imshow(reshape(B(:,6810),28,28));
% subplot (2,3,4), imshow(reshape((pinv(P)*testmatrix(:,6810)),28,28));
% subplot (2,3,2), imshow(reshape(B(:,6811),28,28));
% subplot (2,3,5), imshow(reshape((pinv(P)*testmatrix(:,6811)),28,28));
% subplot (2,3,3), imshow(reshape(B(:,6816),28,28));
% subplot (2,3,6), imshow(reshape((pinv(P)*testmatrix(:,6816)),28,28));
```

```
%%=============== plot eigenvector figures ==========
%[height, width] = size(I);
%neig = 64;
%MTEigenV = zeros(height, width, 1, neig);
%for i = 1 : neig
%   MTEigenV(:, :, 1, i) = reshape(MyRescale(P(i, :), 0, 1), height, width);
%end
%colormap('jet');
%montage(MTEigenV);


end

err = mean(err);

end
```

B Our Attempted Implementation of Neural Network

```
% Neural Network for multi-class classification problem
% Author: Jimmy Lin
% Note that the input hiddenNodes does not include intercept node
% sample input: [300 200 100 50]
% README:  A{1}(4) input of hidden node at layer 1 indexed by 4
%    A - input of each layer node
%    Z - activated value of each node
%    WEIGHTS -
function NN(hiddenNodes, iterations, trainInFile, testInFile, testOutFile)
% TODO:
%  0. fine-tune batch-size, 100 average error as one update
%  1. fine-tune learning rate,
%  2. fine-tune activation function, tan, exp, ...
%
alpha = 0.8;
nHiddenLayers = length(hiddenNodes);
assert(nHiddenLayers > 0)

trainData = csvread(trainInFile);
labels = trainData(:,1);
features = trainData(:,2:size(trainData,2));
nDigits = size(trainData, 1);
nFeatures = size(features, 2);
nClasses = length(unique(labels));

testData = csvread(testInFile);

nLayers = nHiddenLayers+1; % exclude the input layer
% initiailize the weights
WEIGHTS = cell(1, nLayers);
% the weight between input layer and first hidden layer
WEIGHTS{1} = rand(hiddenNodes(1), nFeatures+1);
% weights between hidden layer
for layerIdx = 2:nHiddenLayers,
    WEIGHTS{layerIdx} = rand(hiddenNodes(layerIdx), hiddenNodes(layerIdx-1)+1)
end
% weigths between last hidden layer and output layer
WEIGHTS{nLayers} = randn(nClasses, hiddenNodes(nHiddenLayers)+1);

% training period
fprintf('Start training NN!\n')
for iter = 1:iterations,
    fprintf('Iteration %d Starts\n', iter)
    errors = [];
    for i = 1:nDigits,
        label = labels(i);
        feat = features(i,:)';
        target = zeros(nClasses,1);
        target(label+1) = 1;
        % forward stage
        [A, Z] = forward(feat, hiddenNodes, WEIGHTS);

        % backward stage
        [E, e] = backward(target, hiddenNodes, A, Z, WEIGHTS);
        errors = [errors e];

        % update the weight
        WEIGHTS{1} = WEIGHTS{1} - alpha * E{1} * [1; feat]';
        for layerIdx = 2:nLayers,
            WEIGHTS{layerIdx} = WEIGHTS{layerIdx} - alpha* E{layerIdx} * [1; Z{layerIdx-1}]';
        end
    end
    fprintf('Iteration %d Ends with mean error %f\n', iter, mean(errors))
end
fprintf('Training NN Completes!\n')

% testing period
fprintf('Apply trained NN on testing set!\n')
predictions = [];
for i = 1:size(testData,1),
    feat = testData(i,:)';
    [~, Z] = forward(feat, hiddenNodes, WEIGHTS);
    [~, plabel] = max(Z{nLayers});
    predictions = [predictions; i plabel-1]; % compensate the label offset
end
fprintf('Testing set completes!\n')

predictions
% output to external file
csvwrite(testOutFile, predictions)
end

function [A,Z] = forward(feature, hiddenNodes, WEIGHTS)
        nLayers = length(WEIGHTS);
        nHiddenLayers = length(hiddenNodes);
        % initialize value matrix
        A = cell(1, nLayers);
        Z = cell(1, nLayers);
        A{1} = WEIGHTS{1} * [1; feature];
        Z{1} = tan(A{1});
        % hidden layer
        for layerIdx = 2:nHiddenLayers,
            A{layerIdx} = WEIGHTS{layerIdx} * [1; Z{layerIdx-1}];
            Z{layerIdx} = tan(A{layerIdx});
        end
        % output layer
        A{nLayers} = WEIGHTS{nLayers} * [1; Z{nHiddenLayers}];
        Z{nLayers} = SoftMax(A{nLayers});
end

function [E,error] = backward(target, hiddenNodes, A, Z, WEIGHTS)
        error = 0.0;
        nLayers = length(WEIGHTS);
        nHiddenLayers = length(hiddenNodes);
        E = cell(1, nLayers);
        % output layer
        E{nLayers} = Z{nLayers} - target;
        for layerIdx = nHiddenLayers:-1:1,
            E{layerIdx} = zeros(hiddenNodes(layerIdx),1);
            for j = 1:hiddenNodes(layerIdx),
                E{layerIdx}(j) = sec(A{layerIdx}(j))^2 * (WEIGHTS{layerIdx+1}(:,j)' * E{layerI
            end
        end
        error = error + sum(abs(E{nLayers}));
end

function y = SoftMax(x)
    expv = exp(x-max(x));
    sumExp = sum(expv);
    y = expv / sumExp;
end
```

C Matlab Implementation of Deformation Model

C.1 IDM Classifier

```
%% IDM DEFORMATION MODEL
%% AUTHOR: JIMMY LIN
%% INVOCATION:
%    IDM_Deform('../data/train.csv', '../data/test.csv', '../result/', 1, 1000)
%% USAGE:
%    rawTrainFile - input csv file for training data
%    rawTestFile - input csv file for testing data
%    theStart - the test data instance our prediction starts from
%    theEnd - the test data instance our prediction ends to
function IDM_Deform (rawTrainFile, rawTestFile, outDir, theStart, theEnd)
fprintf('Start reading trainData..\n')
trainData = csvread(rawTrainFile);
train_labels = trainData(:,1);
train_features = trainData(:,2:size(trainData,2));
fprintf('End reading trainData..\n')

fprintf('Start reading testData..\n')
test_features = csvread(rawTestFile);
fprintf('End reading testData..\n')

nTrainDigits = size(trainData, 1);
nTestDigits = size(test_features, 1);
nFeatures = size(train_features, 2);
nClasses = length(unique(train_labels));

filename = strcat(['IDM_' num2str(theStart) '_' num2str(theEnd) '.csv']);
outFile = strcat([outDir filename]);
fid = fopen(outFile, 'w+');
fprintf('Output Filename: %s\n', outFile)

% for parallelism
numWorkers = 14;
LoadEachWorker = nTrainDigits / numWorkers;

for testIdx = theStart:theEnd,
    min_dists = zeros(numWorkers, 1);
    min_imgs = zeros(numWorkers, 1);
    min_classes = zeros(numWorkers, 1);
    test_img = test_features(testIdx,:);

    for workerIdx = 1:numWorkers,
        min_dists(workerIdx) = inf;
        min_imgs(workerIdx) = inf;
        min_classes(workerIdx) = inf;
    end
    parfor workerIdx = 1:numWorkers,
        base = 1+(workerIdx-1)*LoadEachWorker;
        top = workerIdx*LoadEachWorker;
        workLoad = base:top;
        [md, mi, mc] = compute(base, LoadEachWorker, train_features(workLoad,:), ...
                  train_labels(workLoad), test_img);
        min_dists(workerIdx) = md;
        min_imgs(workerIdx) = mi;
        min_classes(workerIdx) = mc;
    end
    [~, index] = min(min_dists);
    min_class = min_classes(index);

    fprintf (fid, '%d,%d\n', testIdx, min_class)
    fprintf ('%d,%d\n', testIdx, min_class)
end
fclose(fid);

end

function [md, mi, mc] = compute (base, Load, train_features, train_labels, test_img)
    min_dist = inf;
    min_img = inf;
    min_class = inf;
    for trainIdx = 1:Load,
        train_img = train_features(trainIdx,:);
        dist = IDM_distance(test_img, train_img);
        if dist < min_dist,
            min_dist = dist;
            min_img = trainIdx + base;
            min_class = train_labels(trainIdx);
        end
    end
    md = min_dist;
    mi = min_img;
    mc = min_class;
end

%% NAME:
%    Computation for IDM distance
%
%% See:
%    Deformation Model for Image Recognition (Algorithm 3)
%
%% INPUT INSTRUCTION
%    A: test image
%    B: reference image
function s = IDM_distance(A, B)
WIDTH = 28;
HEIGHT = 28;
s = 0;
w = 1;
for i = 1:WIDTH,
    for j = 1:HEIGHT,
        min_edist = inf;
        for x = max([1 i-w]):min([WIDTH i+w]),
            for y = max([1 j-w]):min([HEIGHT j+w]),
                dist = norm(A((j-1)*WIDTH+i) - B((y-1)*WIDTH+x));
                if dist < min_edist;
                    min_edist = dist;
                end
            end
        end
        s = s + min_edist;
    end
end
end
```

C.2 Distributed Computation

```
############################################################
##    FILENAME:   getListOfAvailableHosts.py
##    VERSION:    1.0
##    SINCE:      2014-04-27
##    AUTHOR: ##        Jimmy Lin (xl5224) - JimmyLin@utexas.edu
##
############################################################
##    Edited by MacVim
##    Documentation auto-generated by Snippet
############################################################

import HTMLParser
import sys
import os
import datetime

MAX_SERVERS_USED = 50

class TableParser(HTMLParser.HTMLParser):
    def __init__(self):
        HTMLParser.HTMLParser.__init__(self)
        self.in_tr = False
        self.in_td = False
        self.tdIdx = 0
        self.record = []
        self.result = []
        self.isDown = False

    def handle_starttag(self, tag, attrs):
        if tag == 'tr':
            self.in_tr = True
            self.isDown = False
        if self.isDown:
            pass
        elif tag == 'td':
            self.in_td = True

    def handle_data(self, data):
        if self.in_td:
            self.record.append(data)
            self.tdIdx = self.tdIdx + 1
            if data == 'down':
                self.isDown = True
                self.record = []
                self.tdIdx = 0
            if (self.tdIdx == 5 and len(self.record) == 5):
                self.result.append(tuple(self.record))
                self.record = []
                self.tdIdx = 0

    def handle_endtag(self, tag):
        if tag == 'tr':
            self.in_tr = False
        if tag == 'td':
            self.in_td = False

if __name__ == '__main__':

    start = int(sys.argv[1])
    logFile = open(sys.argv[2], 'a+', 0)
    sys.stdout = logFile
    sys.stderr = open('./error.log', 'a+', 0)
```

```
## parse the html
nServerUsed = 0
p = TableParser()
data = ""
for line in sys.stdin:
    data += line
p.feed(data)
## make use of the result
hostsList = p.result
print str(datetime.datetime.now())
hostsList.reverse()
if hostsList[0] is None:
    hostsList = hostsList[1:]
'''
for ho in hostsList:
    print ho
'''

for host, status, upTimes, nUsers, Load in hostsList:
    # double check if there is a down server
    if status == 'down' or nServerUsed >= MAX_SERVERS_USED:
        continue
    nUsers = int(nUsers)
    Load = float(Load)
    if status == 'up' and nUsers <= 1:
        if Load <= 0.05:
            logstr = 'invoke: '+ host+ ' input: '+ str(start)+' end:'+str(start+29)
            #print logstr
            cmd = './autoLogin %d %s' % (start, host)
            cmd += ' & '
            print cmd
            os.system(cmd)
            start += 30
            nServerUsed += 1

logFile.close()
sys.stderr.close()
```

D Matlab Implementation of Ensemble Method

D.1 Ensemble Method by Majority Voting

```
function Ensemble (filelists, outfile)

assert(length(filelists) > 0);

Predictions = [];
for fi = 1:length(filelists),
    file = filelists{fi}
    M = csvread(file, 1, 0);
    if fi == 1,
        Predictions = M(:,2);
    else
        Predictions = [Predictions M(:,2)];
    end
end
size(Predictions)

voting_result = mode (Predictions, 2);
index = 1:size(voting_result, 1);
output = [index' voting_result];

csvwrite(outfile, ['ImageId,Label'])
csvwrite(outfile, output, 1, 0)

end
```

E Invocation of other libraries

E.1 Deep learning Toolbox

```
function DPNN(inTrainFile, inTestFile, outfile)

%inTrainFile = '../data/PCA_train.csv';
%inTestFile = '../data/PCA_test.csv';
%outfile = '../result/PCA_DPNN.csv';

inTrainData = csvread(inTrainFile);
inTestData = csvread(inTestFile);

train_x = double(inTrainData(:, 2:size(inTrainData,2))) / 255.0;
train_y = double(inTrainData(:, 1));
test_x = double(inTestData) / 255.0;

% normalize
[train_x, mu, sigma] = zscore(train_x);
test_x = normalize(test_x, mu, sigma);

%% ex2 neural net with L2 weight decay
rand('state',0);
nn = nnsetup([100 50 10]);

nn.weightPenaltyL2 = 1e-4;  %  L2 weight decay
nn.activation_function = 'sigm';    %  Sigmoid activation function
nn.learningRate = 1;                %  Sigm require a lower learning rate
nn.output              = 'softmax';    %  use softmax output
opts.numepochs =  1;        %  Number of full sweeps through data
opts.batchsize = 100;       %  Take a mean gradient step over this many
opts.plot              = 1;            %  enable plotting

size(train_x)
size(train_y)
nn = nntrain(nn, train_x, train_y, opts);

labels = nnpredict(nn, test_x);

csvwrite(outfile, labels)
```

E.2 Libsvm

```
function NU_SVM (trainFile, testFile, outLabels)

[train_labels, train_instances] = libsvmread(trainFile);

[test_labels, test_instances] = libsvmread(testFile);

model = svmtrain (train_labels, train_instances, '-s 1')

[predicted_labels] = svmpredict(test_labels, test_instances, model);

index = 1:size(test_instances,1);
index = index';

output = [index predicted_labels];

csvwrite(outLabels, ['ImageId' 'Label']);
csvwrite(outLabels, output, 1, 0);

end
```